BS

TECHNICAL REPORT 81-02

Quarterly Technical Report:

Microcomputer Software Engineering, Documentation and Evaluation

LEVEL II

12

James F. Wittmeyer, III

CSM INC.

Computer Systems Management, Inc.

1300 WILSON BOULEVARD, SUITE 102 • ARLINGTON, VIRGINIA 22209

TECHNICAL REPORT 81-02

Quarterly Technical Report:

Microcomputer Software Engineering, Documentation and Evaluation

by

James F. Wittmeyer, III

Computer Systems Management, Inc.
1300 WILSON BOULEVARD, SUITE 102
ARLINGTON, VIRGINIA 22209

TECHNICAL REPORT 81-02

9) QUARTERLY TECHNICAL REPORT 1 Jan - 31 Mar 82

4) MICROCOMPUTER SOFTWARE ENGINEERING, DOCUMENTATION AND EVALUATION. 11 31 Mar 82

14 CSM-82-01

ARPA Order No.:	3829
Contractor:	Computer Systems Management, Inc. 1300 Wilson Boulevard, Suite 102 Arlington, Virginia 22209
Effective Date of Contract:	11/5/79 12 53
Contract Expiration Date:	9/30/81
Contract No.:	15 MDA903-80-C-0155 ARPA
Principal Investigator:	10) Mr. James F. Wittmeyer, III (703) 525-8585
Contract Period Covered:	1/1/81 - 3/31/81
Short Title of Work:	Microcomputer Software Engineering, Documentation and Evaluation

This research was sponsored by the Defense Advanced Research Projects Agency under ARPA Order Number 3829; Contract Number MDA903-80-C-0155; and Monitored by DSS-W. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either express or implied of the Defense Advanced Research Projects Agency or the United States Government.

44758

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 81-02	2. GOVT ACCESSION NO. ----AD-A100703	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Microcomputer Software Engineering, Documentation and Evaluation		5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical 1/1/81 - 3/31/81
		6. PERFORMING ORG. REPORT NUMBER ----
7. AUTHOR(s) James F. Wittmeyer, III		8. CONTRACT OR GRANT NUMBER(s) MDA903-80-C-0155 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Systems Management, Inc. 1300 Wilson Boulevard Arlington, Virginia 22209		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS DARPA Order Number 3829
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE 3/31/81 13. NUMBER OF PAGES 45
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Defense Supply Service-Washington (DSS-W) The Pentagon Washington, D.C.		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Recommended for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Recommended for public release; distribution unlimited.

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Microcomputers; software engineering; software evaluation; documentation.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Microcomputer software for defense applications should be engineered structurally, informed by requirements analyses, documented unconventionally if necessary, and systematically evaluated against an explicit set of performance criteria.

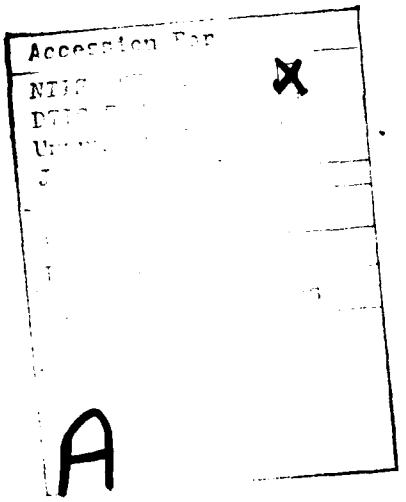
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SUMMARY

This Quarterly Technical Report covers the period from January 1, 1981 to March 31, 1981. The tasks/objectives and/or purposes of the overall project are connected with the design, development, demonstration, documentation, and transfer of advanced command and control (C^2) computer-based systems; this report covers work in the microcomputer software engineering, documentation, and evaluation areas. The technical problems addressed include structured programming, unconventional documentation, and multi-attribute utility-based software evaluation. The general methods employed include software requirements informed structured programming, animated and computer-controlled fiche-based documentation systems, and computer-based software evaluation systems. Technical results include recommendations regarding production rule software selection systems, animated/fiche-based documentation systems, and multi-attribute utility models for software evaluation. Future research will present additional research on microcomputer software and systems design and development.



CONTENTS

FIGURES

TABLES

1.0 INTRODUCTION

Microcomputer "programming is a labor-intensive manufacturing process" (Lewis, 1979). Each year the Department of Defense (DoD) spends billions of dollars on all kinds of macro- and minicomputer software projects, hundreds of millions on microcomputer programming, but relatively little on software engineering, documentation, and evaluation research. This report thus focuses upon several approaches and techniques designed to improve the processes by which we program microcomputers, document microcomputer software, and evaluate software quality and performance--all with reference to DoD research and development needs, requirements, and priorities.

Section 2.0 of this report presents techniques for enhanced microcomputer software engineering. Section 3.0 looks at several useful microcomputer software documentation techniques, while Section 4.0 presents a multi-attribute utility-based model for software evaluation.

2.0 MICROCOMPUTER SOFTWARE ENGINEERING

2.1 Requirements

Ideally before one attempts to build a microcomputer program an effort is made to identify and define the driving functional requirements which together comprise the reason(s) why one attempts to build a problem-solving software system (instead of some other kind of problem-solving system).

At the most basic level are several requirements which are specific context and applications independent; that is, they are relevant to all instances of microcomputer programming regardless of for whom and/or what the software is to be developed.

2.1.1 Response Time - The first is response time. Note that the issue here is not how fast or slowly the system responds to a particular user vis-a-vis a particular task, but how fast or slowly it responds generally. This kind of speed (or slowness) is a function of the software language used and the microcomputer system I/O device times. The figure below, from Barden (1979), presents the total response time for some standard I/O operations.

Operation	I/O Device	Time
Print line of 64 characters.	Teletype	7 seconds
Print line of 64 characters.	IBM Selectric	4 seconds
Print line of 64 characters.	Dot-matrix electrosensitive printer	1–2 seconds
Print line of 64 characters.	Dot-matrix impact printer	1–2 seconds
Display 1024 characters (entire screen.	Video display interface	1 second
Display 1024 characters (entire screen	Crt terminal	2 seconds
Read or write one 100-character record randomly on tape	Audio tape cassette with automatic or manual search	2 minutes
Read or write one 100-character record to next position on tape	Audio tape cassette 30 cps 200 cps	 5 seconds 3 seconds
Read or write one 128-character record randomly on floppy disk	Small floppy disk (5 in) Large floppy disk (8 in	 1/2 second 1/3 second
Read or write one 128-character record to next position on disk	Small floppy disk Large floppy disk	 close to 0

I/O Operation Time

But many operations are non-I/O-oriented, depending instead upon the skill of the programmer and efficiency of the program, which, in turn, depends upon the characteristics of the language used and whether or not the (higher-level) language is compiled or interpreted in operation, as suggested below (Barden, 1979).

Function	Assembly-Language System	Compiler-Language System	Interpreter-Language System
Multiply 1000 numbers of various sizes	1 ms	6 ms	6 s
Divide 1000 numbers of various sizes	1.5 ms	9 ms	9 s
Insert a 20-character string in the middle of 1000 characters of text	7.5 ms	75 ms	10 s
Sort (alphabetize) a list of 100 20-character names	0.1 s	2 s	8 min
Merge 20 names into a list of 100 20-character names	25 ms	0.5 s	2 min
Search 100 20-character randomly ordered names	4 ms	40 ms	1.5 s
Search 100 20-character alphabetized (or otherwise ordered) names	0.4 ms	4 ms	1.5 s

Processing Response Times

2.1.2 Operating Time - Operating time equals I/O time and processing time. But the processing time is always dependent upon the software languages used, the form of the language, and, of course, the efficiency of the programmer, all as suggested below (Barden, 1979).

Application	Assembly-Language System	Compiler-Language System	Interpreter-Language System
Sort and print 1000 names for mailing list; 100 characters/entry, disk system	25 min	25 min	105 min
Generate inventory report of 1000 items; 100 characters/item; disk system	25 min	30 min	41 min
Response time for locating and display of one random account from 2000; disk system	5 s	5 s	30 s

Application Operating Times

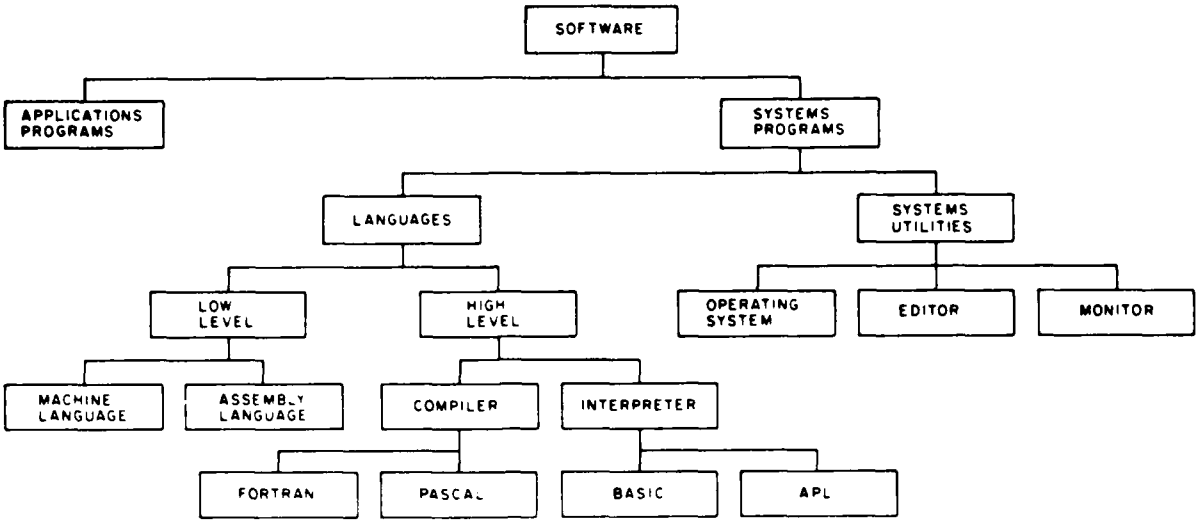
2.1.3 Program Status - Another requirement has to do with the status of the program to be developed. Programs which are fundamentally prototypical or experimental usually bear no resemblance to production (systems or applications) programs. Similarly most programs developed as an initial outgrowth of research and development are iterative in their evolution and should therefore be developed differently from programs intended for wide distribution and use.

2.1.4 Support Requirements - Not unrelated to all of the above are support requirements. Is the program to be transferred for on-line use? Or is it to be used off-line by research and development counterparts? Such questions determine to what extent the software must be self-contained, among other considerations.

2.2 Microcomputer Software Languages

Response and operating time requirements, the status of the program, and support requirements, among many other conceivable requirements, should determine the selection of a software language. Indeed, a set of guidelines regarding the use of one or more languages, of the nature presented below, should be developed and updated frequently in order to ensure the most prudent and practical use of one or another language. In any case the first task is to understand the relationships among the dif-

ferent types of software, as presented below (Frenzel, 1979).



Interrelationships between different types of software

In addition to such relationships are those which surround the requirements, available capabilities, and optimal language selection. (Note that for the purposes of this exemplar exercise substantive requirements are not suggested since they differ from case to case.) For example,

- If response time and operating time is important then one should, assuming programming competence, use compiler rather than interpretive languages for production systems;
- If a system is by definition iterative then interpretive languages should be utilized; and

- If the talent (capability) exists, then machine and assembly languages should be used to maximize the speed of production systems, and so forth.

The point here is that based upon existing empirical studies it is possible to develop sets of guidelines about the selection of software programming languages against explicit requirements. Such guidelines might even be computerized in a developmental reference system which could be used by research and development managers, programmers, and higher-level decision-makers who must make major software investment decisions. Such a production rule system would make systematic a selection process that is now dominated by preference and accessibility.

2.3 Programming Methods

It is difficult to list or define the myriad methods now utilized by programmers. Candidly, most do not have methods which are reproducible (even by themselves) or verifiable. Instead, they usually begin with what they perceive to be the pivotal processing function and they build around it. Most seldom even flow-chart what they intend to program.

Proposed below are several structuring techniques designed specifically to improve microcomputer programming (also see Appendix A for a reprint of an article on structured programming).

In reality they are presented to avoid scenarios like the following (Lewis, 1979):

> *Peter Plodder is slow, methodical, and very meticulous. A mild-mannered, quiet person (with good taste in clothes), he had the irritating habit of issuing long project completion times to his supervisor. Blustering Barton, on the other hand, was a flashy, outspoken superprogrammer who consistently completed his programming assignments ahead of the most optimistic estimates.*
>
> *The Software Division management loved Blust, but hardly knew Peter was alive. Consequently, Blust was granted a six-month leave of absence—a biscuit for his programming accomplishments. A temporary programmer was hired to maintain Blust's code while he was away.*
>
> *Six weeks after Barton embarked on a plane for Africa, his payroll system program failed. The substitute programmer immediately plunged into Barton's program to try to isolate the bug. Perhaps not so surprisingly, he was never able to break into the code. In Blustering Barton's race to produce code, he neglected to write easy-to-understand programs, and his documentation was a mess. In short, only Barton himself could repair the programs he had written.*
>
> *Meanwhile, back at the desk of Peter Plodder business was progressing as usual. Organization and clearly documented programs were his trademark. In fact, Peter was called on to try to find the bug in Barton's payroll program. His time estimate for the debugging task was customarily protracted, but the management had no choice. With Blustering Barton away and the temporary programmer stymied, they had to go with Peter.*
>
> *Eventually the bug was located and corrected, but everyone knew the superprogrammer had stumbled. Summarily, new programming standards were implemented. Peter was invited to teach the other programmers how to write readable code. He showed everyone (including Blustering Barton, when he returned) how to make programs self-documenting. His methodology was adopted as the* only acceptable methodology *to be used throughout the Software Division.*

Sound familiar? Unfortunately, a great many defense research programmers are "flashy, outspoken superprogrammers" who produce jumbled, undocumented software. Consequently, enhancements, modifications, and technology transfer are all made more difficult and much more expensive.

2.3.1 Planning - Structured microcomputer programming is very similar to decision analysis-based problem-solving because it rests upon the principle of problem decomposition (Williams, 1981; Yourdon, 1979; and Ross, et al., 1975). The functions that the program is to perform should inform the decomposition process, and, much like a multi-attribute utility assessment structure, represent functions decomposed to their smallest component units. In this way programmers can adhere to a simple rule of thumb: software solutions should never be more complex than the problems they are intended to solve.

2.3.2 Software Economy - Lewis (1979) bluntly states that programmers should "never write a large program." Instead, he argues convincingly that programmers should write and collect "speedcode modules" that incorporate all of the basic algorithms which the programmer has previously used. Then the modules should be refined onto different microcomputers in different languages.

In a previous report (Wittmeyer, 1980) a design for the development of generic microcomputer-based command and control (C^2) decision and forecasting systems was presented which was based in part upon the use of pre-programmed software modules. It was even suggested that the routine C^2 decision and forecasting systems functions probably numbered less than twenty-five. If this is true then a series of modules (for retrieving

and displaying empirical data, for calculating value, and making inferences, and so forth) could be developed and used over and over again. Similarly, it would be possible to identify and develop modules for information management, training, and generic information display.

Interestingly, most defense software efforts begin from ground-zero and even often ignore previous efforts undertaken by the attending programmer! Clearly a great deal of programming economy can be gained by reviewing existing software and developing reusable software modules.

2.3.3 Software Psychology - All programming methodology must be applied within a particular personnel context; indeed all of the above presumes the existence of highly talented, dedicated programmers who are as knowledgeable about hardware as they are about software. Unfortunately, virtually every projection available today indicates that throughout the 1980s a critical shortage of programmers will persist. We must therefore maximize the output of those programmers which we do employ. Learning, designing, composition, comprehension, testing, debugging, documentation, and modification capabilities must all be evaluated and improved. Perhaps for the first time, serious *programming managers* must pay very special attention to the overall programming environment, the components of which include the physical, social and managerial environments.

2.4 Software Engineering Guidelines and Recommendations

Requirements analyses should precede programming. Requirements should be matched to software characteristics, and then recommendations regarding how to write the software should be generated. In fact, there is no reason why a production rule system such as RITA (Anderson and Gillogy, 1976) could not be used for this purpose. Such a software requirements/software characteristics/programming structure system might be of invaluable use to DARPA researchers specifically and to DoD generally, and might function as follows: users could input requirements consisting of operating and response time requirements, program status requirements, support requirements, among any number of other requirements and the computer system, from a knowledge base consisting of software characteristics (updated continually), would then make recommendations regarding optimal programming efficiency in structured pseudocode supplemented by graphic flowcharts of same. It might also suggest the use of pre-programmed software modules about which it has been given detailed information. The information about software form and language characteristics could be consensus "expert" data or data gleaned from empirical experiences with the software; regardless, the system would enable microcomputer programmers to benefit from existing experience with and information about microcomputer software and thereby generate more efficient code.

This idea is aimed at supporting the microcomputer programmer; more advanced ideas may very well result in computer generated software in the not too distant future.

3.0 MICROCOMPUTER SOFTWARE DOCUMENTATION

Without effective documentation software dies a slow and painful death. Along the way software research progress is encumbered, demonstrations are complicated, and technology transfer is undermined. Interestingly, while the disasterous effects of non-existent or poor documentation are widely verifiable, few are willing to allocate resources aimed at improving documentation techniques. The reason is simple: documentation and documentation research are relatively boring analytical subtasks connected with the potentially exciting design and development of microcomputer-based systems.

At the same time, some effort has been made to define and improve documentation (see Appendix B), and given the progress recently made in voice input/output system development, video technology, interactive graphics technology, and computer-controlled microfiche systems development, it is now possible to experiment with the development of several variations of unconventional documentation not possible just five years ago. For example, systems should be programmed to introduce and explain themselves in a manner not unlike that which is used by manufacturing vendors. Such demonstrations could be of invaluable help to those who must convinve others that what they have developed may be of real use. Documentation should

also be transformed from the inanimate to the animate. Computer-generated system specifications and functional descriptions can be of immense transfer use, as can on-line users manuals. Similarly, films of documentation can also help to bridge the gap between the developer and the user. Here computer-controlled fiche could be used to minimize cost, time delay, and obsolescence. Similarly large screen display systems could be used to present complicated documentation "blueprints" to large audiences and program conversion teams and groups. Self-documentation and automatic flowcharting systems should also be developed. Indeed, the approach now taken by MIT regarding the development of videodisc-based training systems could be used to develop videodisc-based documentation systems.

4.0 MICROCOMPUTER SOFTWARE EVALUATION

Evaluating microcomputer software can be exasperating. In the 1970s--with a good deal of DARPA support, however, a methodology was developed to assist decision makers with complicated evaluation problems. The methodology was subsequently incarnated as a microcomputer program called "EVAL."

4.1 The Evaluation Methodology

At the core of EVAL lies an evaluation methodology known as multi-attribute utility theory (MAUT). Developed at the University fo Southern California by Ward Edwards, MAUT "can spell out explicitly what the values of (a) decision maker are, ...and show how much they differ" (Edwards, 1977). The values themselves are determined against a set of evaluative criteria (or attributes) which are arranged hierarchically in a MAU model. The construction of a MAU model thus begins with "the overall top-level criterion for which a comparative evaluation score is desired. That factor is successively decomposed into its component criteria in descending levels of the hierarchy such that each successive lower-level criterion is more specific than those at the preceding level..." (Allardyce, et al., 1979). The criteria are then weighted in terms of their importance and then the decision maker scores the objects under evaluation against all of the criteria.

4.2 EVAL

EVAL is a generic APL program which currently resides on an IBM 5110. Through EVAL, a decision-maker can create, store, retrieve, and refine MAU models interactively. A Typical MAU model appears below.

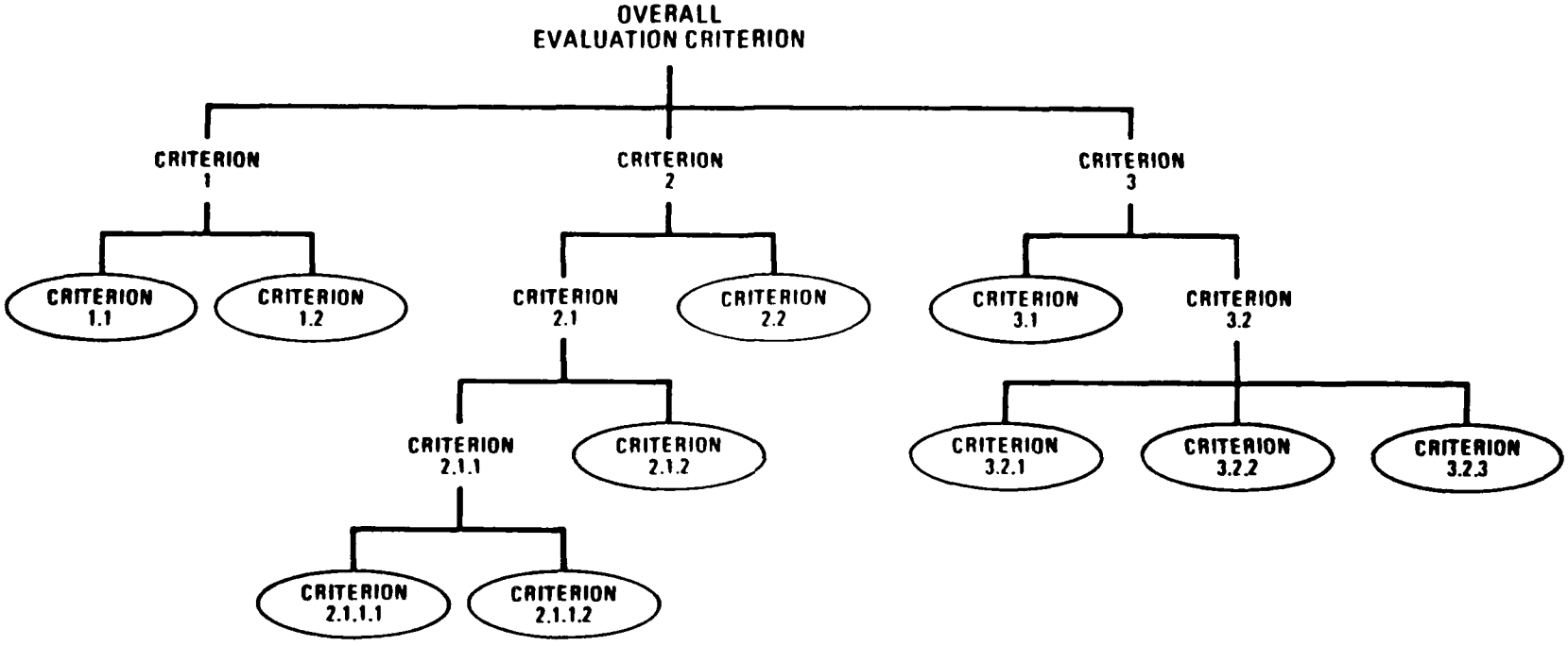
The use of EVAL is fixed according to the following elements (see Allardyce, et al., 1979):

- The Evaluation Problem:
  - A label identifying the problem;
- Criteria:
  - A set of evaluative criteria decomposed into component criteria;
- Alternatives:
  - A list of (labeled) alternatives which the decision maker must evaluate;
- Utility Scores:
  - A list of scores (expressed as a number between 0 and 1) representing the relative utility of each alternative evaluated with respect to each (bottom-level) criterion;
- Relative Importance Weights:
  - Weights which describe the relative importance of lower-level criteria. All criteria (expect for the overall /top-level/ criterion) are assigned importance weights;

- Data Identification Numbers (DINs):
  - These are assigned to each criterion and describe how the criteria are related. This numerical labeling process is shown in the following figure. (For example, the sub-criteria of criterion 1 have data identification numbers 1.1 and 1.2.)

The above input specifications can then be processed to yield the following results:

- Overall Results:
  - The overall value or "worth" associated with each alternative obtained by weighting and adding the value scores assigned to the bottom-level criteria, aggregating from the bottom to the top;
- Normalized Weights:
  - A set of vectors corresponding to the relative criteria importance weights;
- Intermediate Results:
  - Values assigned to any of the intermediate criteria as they contribute to the overall results;
- Cumulative Weights:
  - Weights corresponding to the relative criteria importance weights calculated as follows: "top-level criteria comprising the overall evaluation have cumulative weights equal to their normalized weights. At the next lower level, the criteria are assigned a cumulative weight computed by multiplying the normalized weight by the cumulative weight of the factor to which it is attached, and dividing the product by 100. This process is continued down



THE FORMAT OF A MULTI-ATTRIBUTE UTILITY ASSESSMENT MODEL

through the structure until all criteria have been assigned cumulative weights. The cumulative weight (CUMWT) indicates the relative importance of the criterion to the overall evaluation" (Allardyce, et al., 1979);

- Sensitivity Analysis:
  - The user identifies a single criterion of interest and assigns the maximum and minimum cumulative weights that it may assume. EVAL then varies the cumulative weight of the criterion in increments of one-tenth of the difference between the maximum and minimum weights, while the other weights in the model maintain their previously assigned proportional relationships with one another. Generally, the alternative that receives the highest overall utility will change as the criterion weight is incremented from W_{min} to W_{max}. The changes are referred to as threshold points, as shown below. (Note that the alternative having the highest value is designated with an asterisk.)

1.2 PERFORMANCE CURRENT CUMWT: 55.00

WEIGHT	A	B	C	D	E
.0	63	54	50	68	74*
10.0	63	56	52	66	73*
20.0	64	59	54	64	72*
30.0	64	62	56	62	71*
40.0	65	64	58	60	70*
50.0	65	67	60	58	69*
60.0	66	70*	62	56	68
70.0	66	72*	63	54	67
80.0	67	75*	65	52	66
90.0	67	78*	67	49	65
100.0	68	81*	69	47	64

SENSITIVITY ANALYSIS

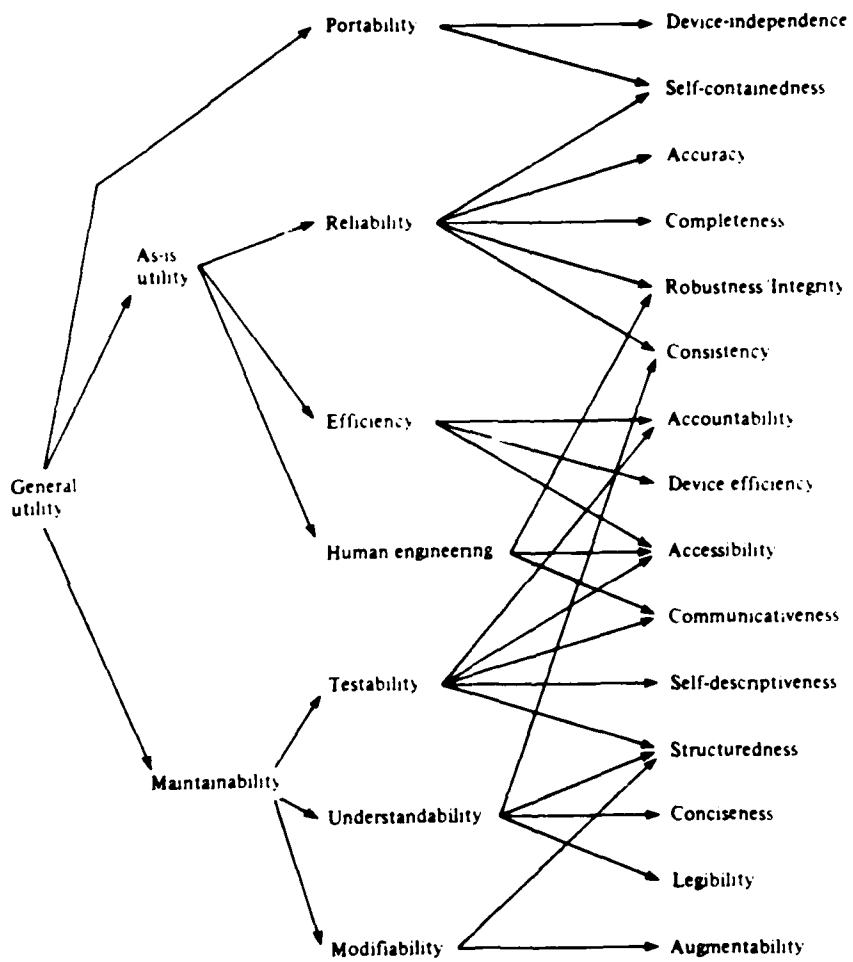
4.3 A Microcomputer Software Evaluation System

Boehm, et al. (1977) have developed a software characteristics tree which has been converted by CSM into a multi-attribute utility model for the evaluation of software quality. Like all EVAL models it is changeable; nevertheless, we think it is probably very useful as is. Also like all EVAL models the criteria have been defined (according to Boehm, et al., 1977):

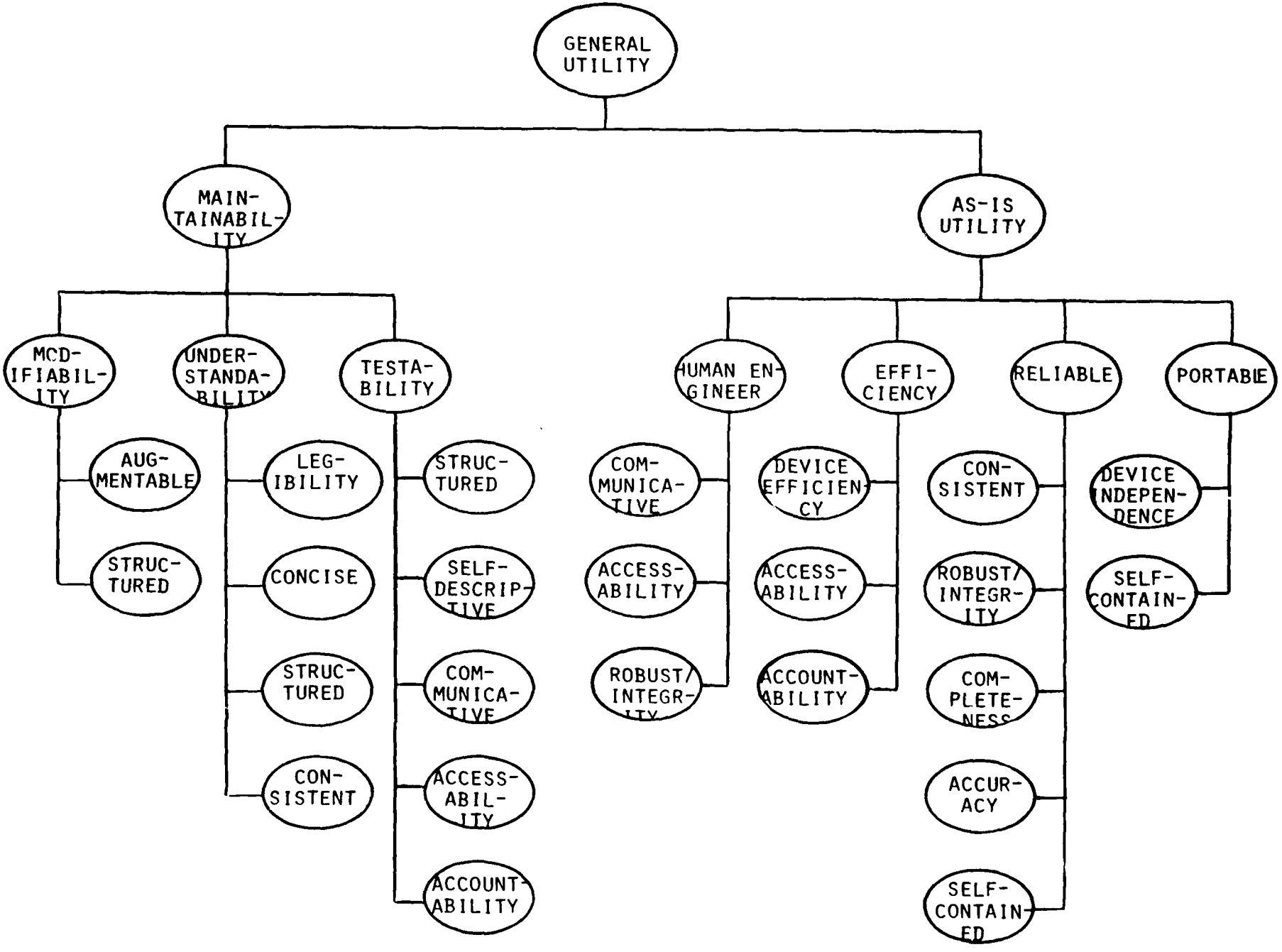
- Accessibility: Extent to which code facilitates use of its parts;
- Accountability: Extent to which code can be measured;
- Accuracy: Extent to which the output produced by code are sufficiently precise to satisfy their intended use;
- Augmentability: Extent to which code can be expanded in computations functions, or data storage requirements;
- Availability: Degree to which a system of resource is ready to process data. Availability. MTBF/(MTBF + MTTR);
- Communicativeness: Extent to which code facilitates the specifications of inputs and provides outputs whose form and content are easy to assimilate;
- Completeness: Extent to which all parts of code are present and developed;
- Conciseness: Extent to which excessive information is not present;
- Consistency: Extent to which code contains uniform notation, terminology, and symbology within itself, and external consistency to the extent that the content is traceable to the requirements;

- Device independence: Extent to which code can be executed on computer hardware configurations other than its current one;
- Efficiency: Extent to which code fulfills its purpose without wasting resources;
- Human engineering: Extent to which code fulfills its purpose without wasting users' time and energy or degrading their morale;
- Legibility: Extent to which function is easily discerned by reading codes;
- Maintainability: Extent to which code facilitates updating;
- Modifiability: Extent to which code facilitates the incorporation of changes;
- Portability: Extent to which code can be operated easily and well on computer configurations other than its current one;
- Reliability: Probability that an item (device or program, system) will function without failure over a specified time period or amount of usage;
- Robustness: Extent to which code can continue to perform despite a violation of the assumptions in its specifications;
- Self-containedness: Extent to which code performs its explicit and implicit functions within itself;
- Self-descriptiveness: Extent to which reader of code can determine its objectives, assumptions, constraints, inputs, outputs, components, and revision status;
- Testability: Extent to which code facilitates establishment of verification criteria and supports evaluation of its performance;
- Understandability: Extent to which purpose of code is understandable to reader; and
- Usability: Extent to ...ich code is reliable, efficient, and human-engineered.

As presented previously (Wittmeyer, 1980), the Boehm, et al. (1977) software characteristics tree is as follows:



When this tree is arranged hierarchically in a multi-attribute utility model, it appears as follows:


GENERAL UTILITY
MAIN-TAINABIL-ITY
AS-IS UTILITY
MOD-IFIABIL-ITY
UNDER-STANDA-BILITY
TESTA-BILITY
HUMAN EN-GINEER
EFFI-CIENCY
RELIABLE
PORTABLE
AUG-MENTABLE
STRUC-TURED
LEG-IBILITY
CONCISE
STRUC-TURED
CON-SISTENT
STRUC-TURED
SELF-DESCRIP-TIVE
COM-MUNICA-TIVE
ACCESS-ABIL-ITY
ACCOUNT-ABILITY
COM-MUNICA-TIVE
ACCESS-ABILITY
ROBUST/ INTEGR-ITY
DEVICE EFFICIEN-CY
ACCESS-ABILITY
ACCOUNT-ABILITY
CON-SISTENT
ROBUST/ INTEGR-ITY
COM-PLETE-NESS
ACCUR-ACY
SELF-CONTAIN-ED
DEVICE INDEPEN-DENCE
SELF-CONTAIN-ED

5.0 CONCLUSION

This report suggests the following:

- A set of programming standards including especially structured programming techniques, should be developed and applied to DARPA/DSO/CTD projects;
- A computer-based production rule system should be developed which would enable programmers to input programming requirements and receive guidance and recommendations regarding how to program, which language to use, and the like;
- Software documentation should be animated via several media including computer-based fiche and videodisc systems;
- Automatic flowcharting and self-descriptive software systems should be developed and tested for their documentation effectiveness in technology transfer contexts; and
- Multi-attribute utility-based models of software quality should be developed and exercised in order to assess existing and improve on-going software projects.

6.0 REFERENCES

Allardyce, L.B.; Amey, D.M.; Feuerwerger, P.H.; and Gulick, R.M. Documentation of Decision Aiding Software: EVAL Functional Description. McLean, Virginia: Decisions and Designs, Inc., November 1979.

Anderson, R.H. and Gillogy, J.J. Rand Intelligent Terminal Agent (RITA): Design Philosophy. Santa Monica, California: The Rand Corporation, February 1976.

_____. Rand Intelligent Terminal Agent (RITA): Reference Manual. Santa Monica, California: The Rand Corporation, December 1976.

Edwards, W. "How to Use Multiattribute Utility Measurement for Social Decisionmaking." IEEE Transactions on Systems, Man, and Cybernetics, Volume SMC-7, Number 5, May 1977, pp. 326-340.

Barden, W., Jr. Microcomputers for Business Applications. Indianapolis, Indiana: Howard W. Sams and Co., 1979.

Freeman, P. and Wasserman, A.I., eds. Tutorial on Software Design Techniques. IEEE Computer Society, 1980.

Frenzel, L.E., Jr. "Understanding Personal Computer Software." On Computing, Winter, 1979.

Higgins, D.A. Program Design and Construction. Englewood Cliffs, New Jersey: Prentice Hall, 1979.

Howard, J. "What is Good Documentation?" Byte, March 1981.

Lewis, T. . Software Engineering for Micros. Rochelle Park, New Jersey: Hayden Book Company, Inc., 1979.

Ross, D.T.; Goodenough, J.B.; Irvine, C.A. "Software Engineering." Computer. IEEE, May 1975.

Shneiderman, Ben. Software Psychology. Cambridge, Massachusetts: Winthrop, 1980.

Waite, M. and Pardee, M. Microcomputer Primer. Indianapolis, Indiana: Howard W. Sams & Co., 1980.

Williams, G. "Structured Programming and Structured Flowcharts." Byte, March 1981.

_____. "Applied Structured Programming." Program Design. Blaise Liffeck, ed. Peterborough, New Hampshire: Byte Books, 1978.

Wittmeyer, J.F. Defense Microcomputing in the 1980s. Computer Systems Management Technical Report, December 1980.

Yourdon, E.N., ed. Classics in Software Engineering. New York, New York: Yourdon Press, 1979.

APPENDIX A

"Structured Programming and Structured Flowcharts"

by

Gregg Williams

Structured Programming and Structured Flowcharts

Gregg Williams, Editor
BYTE
POB 372
Hancock NH 03449

Structured programming—that phrase, unfamiliar to me and, I assume, to most people several years ago—is now endowed with such magical powers that most books on programming include it somewhere in their titles.

But what is structured programming? Most of us feel that it is probably good for us, like getting regular exercise or brushing our teeth after each meal. You may also think it's too complicated (not true), that it slows down programming (wrong, it usually speeds it up), or that it cannot be done unless your computer runs a language like Pascal or ALGOL (wrong again).

Simply put, structured programming is a set of techniques that makes programs easier to write, easier to understand, easier to fix, and easier to change. These techniques are simple and general and can be adapted to any computer language that has a *goto* statement—that includes BASIC, assembly language, FORTRAN, and COBOL. The purpose of this article is to show you a new form of notation that will help you write structured programs. But first, let's review structured programming.

The Elements of Structured Programming

A structured program is like a set of notes written in outline form. The headings accompanied by Roman numerals—I, II, III, and so on—provide the overall organization. Each Roman numeral topic is broken into several component topics (A, B, and C, for example) and each of these is subdivided further (1, 2, 3, ...) and further (a, b, c, ...) as needed. Table 1 shows a problem and its solution written in this outline form.

The above example demonstrates a process known as *decomposition*: breaking a task (problem) into its subtasks. This process represents the *most* important concept in structured programming, ie: that a problem can be solved by repeatedly breaking it into subproblems, until *every* subproblem can be solved. If you plan this decomposition before you try to write it out in the narrow, precise, and time-consuming syntax of the target language (ie: the programming language you use to solve the problem), you will have a better chance of getting your program right the first time.

It has been *mathematically proven* that any program can be written using three basic constructs.

But how do you decide which way to break the problem into subproblems? Common sense helps. Ask yourself, "What sequence of actions and decisions would I have to make if I were doing this without a computer?"

The rest of the answer comes from the literature of structured programming. It has been mathematically proven that *any* program can be written using three basic patterns, called *programming constructs* (or simply *constructs*): sequence, *if...then...else*, and *while...do*. The first construct, sequence, gives you the basic capability of breaking a task into a set of subtasks that accomplish the main task when executed sequentially.

The second construct, *if...then...else*, performs one of two subtasks, depending on the truth or falsity of a stated condition. An everyday example of this construct is given in the following sentence: "If it is raining outside, I will take my umbrella with me; if it is not, I will leave the umbrella at home."

The third and least familiar construct, *while...do*, is actually a generalized *do-loop* that repeats a set of actions (called the *body* of the loop) while a stated condition is true. You use this construct when making iced tea from a mix: "As long as (while) the mix is not completely dissolved, I will continue to stir it."

If you combine lines of code in the three ways described above, the resulting program is said to be *structured*. In most languages (BASIC, for example) you will still use *goto* statements, but they will be *restricted* to carrying your program to specific points, ie: the beginnings and ends of tasks or subtasks. Each module (subtask) in a structured program has a property known as "one-in, one-out"; that is, there is only one entrance and one exit from these modules, and no module will ever jump into the middle of another one. Instead of being like a plate of spaghetti, a program is more like a string of pearls (with each pearl containing another, smaller string of pearls, and so on); each module has a definite and unchanging position on the string. When such regularity can be counted on, existing modules can be changed or deleted, and entirely new modules can be add-

Structured Programming: A Qualification

About a year ago, I thought that structured programming was the ultimate tool in the analysis, design, and implementation of a computer program. I had read several books on the subject, browsed through a great many more, and successfully applied the techniques to real-world problems. Many books spoke of structured design, but I saw the concept as simply the same structured programming tools applied to the earlier process of program design—that is, of transforming a situation to be solved into a set of programs that will accomplish the task. I was more wrong than right.

Through my experience with a particular programming project, I suddenly recognized a major point that I had formerly not comprehended: that structured programming does not encompass the entire process of programming. The [illegible] begins with some sort of description or specification of the program to be written. With small programs (like the kind we spend most of our time writing), this is usually enough. But as the problem gets bigger (and perhaps more ill-defined), more and more crucial design decisions must be made before you divide the problem into programs.

I also learned that certain design decisions within a given program are overlooked by the main ideas of structured programming. Structured programming is a literal-minded discipline that deals exclusively with the orderly disassembly of a problem into the series of program statements that solves it. It does this while assuming several givens: the overall algorithm to be used (eg: bubble sort or heapsort), the data structures used (eg: linked lists, arrays, or binary trees), and implementation details (eg: sequential or random-access files, the packing of one or two characters per byte). These details, which may have a tremendous effect on the quality of the program (in such aspects as size, speed, readability, and maintainability), are factors that are evaluated and weighed in the design process.

The purpose of these paragraphs is two-fold: first, to affirm that the techniques described in this article can make a significant improvement in your skills as a programmer and that they are sufficient for many programs; and, second, to emphasize that the quality of a program can often be greatly improved by attention to the design decisions that are made in the early stages of analyzing the program design. I am including a list of particularly helpful books and articles in the references at the end of this article.

ed without problems caused by unexpected module interaction.

That is the theory of structured programming—now for putting it into practice. Figures 1 thru 3 show the three constructs (sequence, *if... then...else*, and *while...do*) in standard flowchart form and as BASIC code. (For a more detailed look at writing structured programs in BASIC, see "Applied Structured Programming," listed in the references. This article appears in an anthology that contains several other good articles on program decomposition—sometimes called *top-down design* or *programming by stepwise refinement*—and structured programming.)

The Origins of a New Notation

When I got my first job as a commercial programmer, I realized that I was going to have to write longer programs than I had previously written. This prompted me to adapt structured programming techniques to my work in BASIC, COBOL, and RPG II. (As it turned out, my longest program was a 35-page COBOL program that grew to 75 pages without going out of control. I could not have done this without the rigorous use of structured programming techniques.)

As my programs grew larger, I became dissatisfied with the methods I used to plan my programs. Conventional flowcharts obscured the structure of my programs. Nassi-Schneiderman charts and Warnier-Orr diagrams were unsatisfactory for other reasons.

The best solution offered in structured programming texts was *structured pseudocode*, an informally written Pascal-like "program" that uses terse English phrases to describe the program. Listing 1 shows the structured pseudocode for the program outlined in table 1b. I used structured pseudocode extensively to outline programs but found that the details of the resulting pseudocode often obscured the overall design of the program.

In retrospect, I can see that I wanted a design notation that could do the following:

- Completely describe the algorithm to be programmed
- Provide overview and detailed documentation that was easy to read
- Not need to be redrawn every time

Problem: Given a numeric array V with N elements, find the largest element, MAXV, and its index, MAXINDEX. These variables are related as follows:

- 1 < MAXINDEX < N
- MAXV = V (MAXINDEX)
- MAXV is the largest value in V(1), V(2)... V(N)

Solution:

I. Set problem up
   A. Set MAXVAL = -9×10^{20}
   B. Set MAXINDEX = 0
   C. Set INDEX = 1
II. Find largest element
   A. Set up a loop that increments the variable INDEX from the beginning to the end of the array V.
   For each value of INDEX
      1. Compare the current array value (V (INDEX)) to MAXVAL
         a. if MAXVAL is equal or larger, do nothing
         b. if MAXVAL is smaller, replace MAXVAL with the current array value and MAXINDEX with the current index (the value of INDEX)
III. Print the largest element (MAXVAL) and its index (MAXINDEX)

Table 1: *A problem and its solution in outline form. The common outline form used for summarizing a body of material can also be used to give structure to the emerging design of a program. Table 1a gives a statement of the problem and table 1b gives its solution in outline form.*

Listing 1: *A structured pseudocode solution of the FINDMAX problem given in the text and in table 1. Structured pseudocode is a terse, informal, Pascal-like program that helps the user design a program before writing it in a formal programming language.*

```
Program FINDMAX:
Initialize system variables (MAXV = -9 x 10^20, MAXINDEX = 0, INDEX = 1)
While INDEX ≤ N
  find value of current array element ( CURRV = V (INDEX) );
  if current array element (CURRV) > maximum element so far (MAXV)
    new maximum element = current element
    new maximum index = current index ( MAXINDEX = INDEX )
  endif
  increment INDEX by 1
endwhile
print MAXV, MAXINDEX
(end of program)
```

Listing 2: *A BASIC implementation of the FINDMAX problem from table 1. In this program, the variable MAXINDEX has been shortened to MINDEX to distinguish it from the variable MAXV. This program is written in TRS-80 Model I Level II BASIC, and it will run on other computers that use Microsoft BASIC.*

```
100 :
110 REM                  PROGRAM FINDMAX
120 :
130 REM     THIS PROGRAM TAKES AN ARRAY OF NUMBERS, V, AND
140 REM   FINDS THE LARGEST ELEMENT, MAXV, AND ITS INDEX,
150 REM   MAXINDEX, SUCH THAT:
160 REM                MAXV = V (MAXINDEX)
170 :
180 REM     (FOR THE PURPOSES OF ILLUSTRATION, WE WILL ASSUME
190 REM   THAT THE DATA IS ALREADY IN THE ARRAY V.)
200 :
210 :
220 REM ================== MAIN PROGRAM ======================
230 :
240 DIM V(12)
250 GOSUB 800: REM   --NOT PART OF ALGORITHM IN FIGURE 6; THIS
260 REM                SUBROUTINE ENTERS DATA INTO ARRAY V
270 :
280 REM ---------- BOX 1: INITIALIZATION ROUTINE ---------------
290 :
300 MAXV = -9 * 10^20
310 MINDEX = 0
320 INDEX = 1
330 :
340 REM ---------- BOX 2: FIND LARGEST VALUE -------------------
350 :
360 REM -- (BEGINNING OF WHILE...DO LOOP)
370 IF INDEX   N THEN 520
380     CURRV = V (INDEX)
390 :
400     IF CURRV   MAXV THEN 440
410          MAXV = CURRV: REM -- (THIS PART EXECUTED IF FALSE)
420          MINDEX = INDEX
430 :
440     INDEX = INDEX + 1
450 :
460 REM -- (JUMP TO BEGINNING OF WHILE...DO LOOP)
470 GOTO 370
480 :
490 :
500 REM ---------- BOX 3: PRINT FINAL VALUES -------------------
510  :
520 PRINT:  PRINT "THE LARGEST VALUE IN THE  V  ARRAY IS:"
530 PRINT "          V("; MINDEX ; ") = "; MAXV
540 PRINT
550 :
560 END
570 REM ============== END OF MAIN PROGRAM ====================
760 :
770 :
780 REM ---------- SUBROUTINE TO FILL V ARRAY ------------------
790 :
800 DATA 12: REM -- (NUMBER OF ITEMS TO BE READ IN)
810 DATA 1, 15, -28, 3.24, -17.92, 0, 5, 1, 0, 21.4, -205, 17
820 READ N
830 FOR I=1 TO N: READ V(I): NEXT I
840 RETURN
```

a change was made in the flowchart
- Use a minimum of unfamiliar notation
- Be visually pleasing

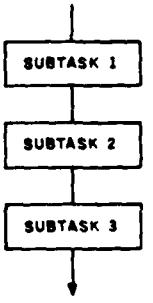
This *structured flowchart* notation, which I developed over a period of several years, meets these criteria.

Basic Constructs in Structured Flowcharting

According to the tenets of structured programming, any program can be expressed as a combination of four basic building blocks. These are sequence, *if...then...else*, *while...do*, and decomposition. (The first three constructs, described in conventional flowcharts in figures 1a thru 3a, are given in structured flowcharts in figures 4a, 4b, and 4c, respectively.)

The sequence construct (figure 4a) is identical for both conventional and structured flowcharts; however, a later construct, decomposition, will distinguish the structured flowchart sequence construct from its conventional counterpart.

(a)

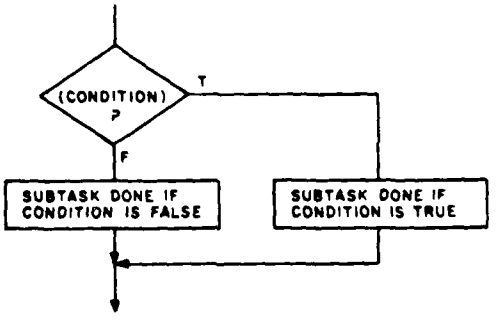


(b)

```
100  (BASIC statement for subtask 1)
110  (BASIC statement for subtask 2)
120  (BASIC statement for subtask 3)
```

Figure 1: *Sequence as a control structure. Figure 1a shows how a linear sequence of subtasks is drawn using conventional flowchart notation. Figure 1b shows the equivalent sequence as a series of BASIC lines.*

(a)

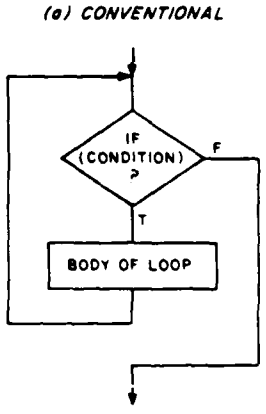


(a) CONVENTIONAL

(b)

```
100  IF (condition) THEN 200
120  (BASIC statements for subtask
     done if condition is false)
      .
      .
190  GOTO 300
200  (BASIC statements for subtask done if
     condition is true)
      .
      .
299  (last statement of "true" subtask)
300  (first statement of next construct)
```

Figure 2: *The if...then...else construct as a control structure. Figure 2a shows the conventional notation for this construct, while figure 2b shows the BASIC equivalent.*



(b)

```
100  IF (opposite of condition) THEN 300
110  (BASIC statements for body of loop,
       done if condition is true
 .
 .
 .
299  GOTO 100
300  (first statement of next construct)
```

Figure 3: *The* while...do *loop as a control structure. Figure 3a shows the* while...do *loop in conventional flowchart notation. Figure 3b shows the equivalent loop in BASIC code.*

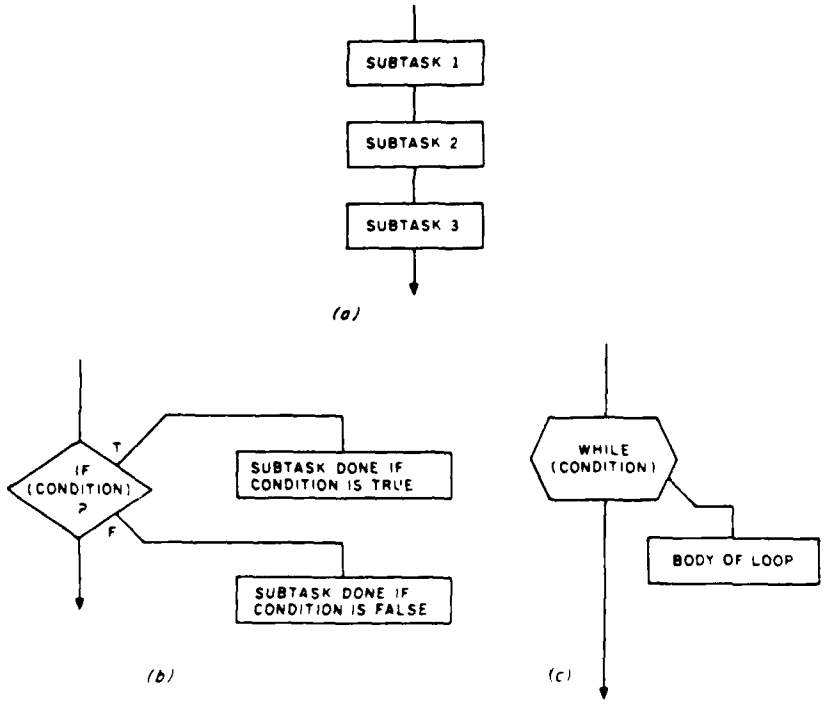


Figure 4: *The basic structured flowchart notations. Figure 4a shows the structured flowchart notation for a sequence of tasks; it is equivalent to the flowchart of figure 1a. Figure 4b shows the structured flowchart notation for the* if...then...else *construct (equivalent to figure 2a); note that it is the placement of the letters T and F (for* true *and* false*) that determines the conditions under which a given subtask is performed. Figure 4c shows the structured flowchart notation for the* while...do *construct (equivalent to figure 3a); the diagonal line leading down indicates that the condition (in the hexagon) is performed before the body of the loop.*

The *if...then...else* construct is fairly straightforward in the conventional flowchart (figure 2a). In the structured flowchart version (figure 4b), the boxes to be performed are to the right of the decision diamond, with the understanding that only one of the two boxes will be performed based on the value of the condition in the diamond. If the "else" side of the construct is not needed, the box labeled F is eliminated. In this case, if the condition does not evaluate to *true*, no action is performed, and control continues with the next construct following the decision diamond.

The notation for the *while...do* construct is not as easily derived. The conventional flowchart cannot directly express this kind of loop; it must use a decision diamond and an external loop (figure 3a). The structured flowchart version (figure 4c) introduces a new symbol, a hexagon. (Actually, the hexagon is used to denote one of several kinds of loop structures; the word *while* makes this a *while...do* loop.) The box connected below and to the right of the hexagon is performed as long as the condition

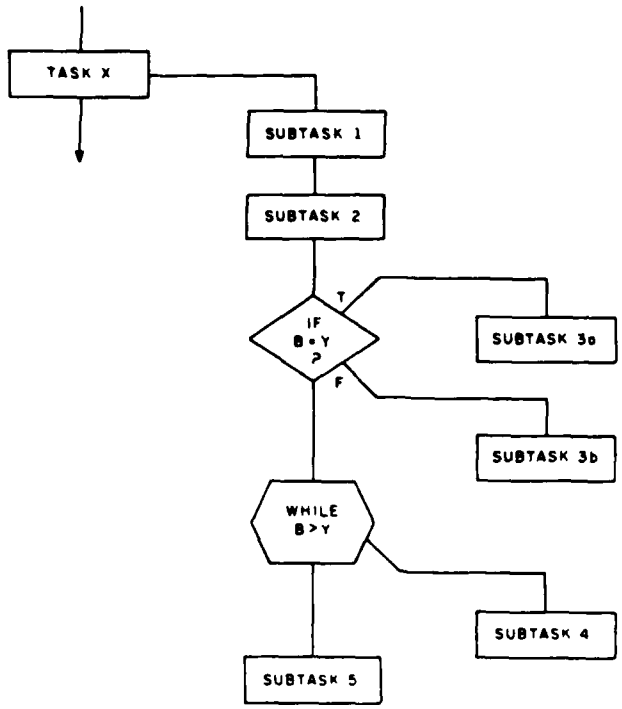


Figure 5: *Example of the subdivision of a task. A central rule of structured flowcharts is that any box can be broken into multiple boxes that represent the necessary subtasks. Here, task X is broken into five subtasks executed in top-to-bottom order. Subtasks 1, 2, and 5 are simple subtasks. Subtask 3 is an* if...then...else *construct. Subtask 4 is a* while...do *loop*

listed in the hexagon is true. The condition is performed first (denoted by the position of the hexagon being spatially *above* the box being performed); this allows the possibility of the body of the loop being performed zero times if the condition is initially false.

The fourth and pivotal construct of this programming notation, decomposition, can best be stated as a rule: *any box representing a task can be broken into multiple boxes that represent the necessary subtasks.* The subtasks may be rectangular boxes that represent simple tasks, or they may be any other valid structured flowchart construct (*if...then...else*, *while...do*, etc). They are written top to bottom in the order of performance, with the line denoting program flow entering each subtask box from its top and exiting from the bottom.

Figure 5 illustrates the above construct. Task X is composed of five subtasks performed in numeric sequence. Tasks 1, 2, and 5 are simple subtasks. Subtask 3 is an *if...then...else* construct that allows either subtask 3a or subtask 3b to be per-

formed. Subtask 4 is performed as long as the condition within the hexagon (B>Y) is true. Of course, any subtask box may be further divided into its component subtasks.

Since any box can be broken into component subtasks, you can now see how this notation is used to design a program. The boxes in the leftmost column give the overall design of the program; boxes are then expanded to the right as each box (task) is divided into boxes representing the appropriate combination of subtasks. As a result, you can scan any one of several of the leftmost column of boxes for an overview of varying depths of the program design, or you can study the implementation of any major or minor subtask by concentrating on only the boxes and control structures growing to the right of the given subtask.

An Example

The following example will illustrate the process of developing a program using structured flowcharts. Using the example of table 1a, suppose you are given an array of N numbers, V(1), V(2),...V(N), and have to find the index value MAXINDEX such that the largest value in the V array is MAXV=V(MAXINDEX). The entire structured flowchart for this problem is given in figure 6.

Cover the right three-fourths of the flowchart so that only the subtasks numbered 1, 2, and 3 are visible. This is what the "first pass" of the flowcharting effort should look like. Subtask 1 is the initialization of the problem. Subtask 2 is the determination of MAXINDEX and MAXV. Subtask 3 is the printing of these two values. Since the task in subtask 3 is simple enough to be directly accomplished in the target language (for example, BASIC), it need not be subdivided.

Subtasks 1 and 2 are developed concurrently. Subtask 2 is basically a loop that examines V(1), V(2),...V(N) in turn, keeping the appropriate values for MAXV and MAXINDEX for the I elements encountered thus far. The values of MAXV, MAXINDEX, and INDEX must be set (as is done in subtasks 1.1, 1.2, and 1.3). Note that this loop could have been done more easily using a do-loop; other optimizations could also have been made, but this example is given for the purposes of illustration only.

The main work for each element is done as subtask 2.1.2: if the current V element being examined (ie:CURRV) is greater than the maximum V element so far, MAXV and MAXINDEX are set to the current array and index values, respectively. These subtasks, numbered 2.1.2.1 and 2.1.2.2, are performed only when the relationship given in the diamond of 2.1.2 is true.

Once the structured flowchart has reached the level of detail shown in figure 6, most of the design considerations have been conceived and perfected; it is then a simple task to translate the program into BASIC (see listing 2) or any other general-purpose computer language. The benefits are more pronounced when used with a larger program. If a structured flowchart is subdivided to the right until each box represents a task that can be directly coded in the target language, you will catch most of the "oops, I forgot to..." insertions and changes that programmers generally think of *after* they have started coding the program.

Other Control Structures

Although the three constructs discussed so far are sufficient for writing any program, it is not always convenient to use only these constructs. Other control structures can be devised for the convenience of the programmer. For example, boxes 1.3,

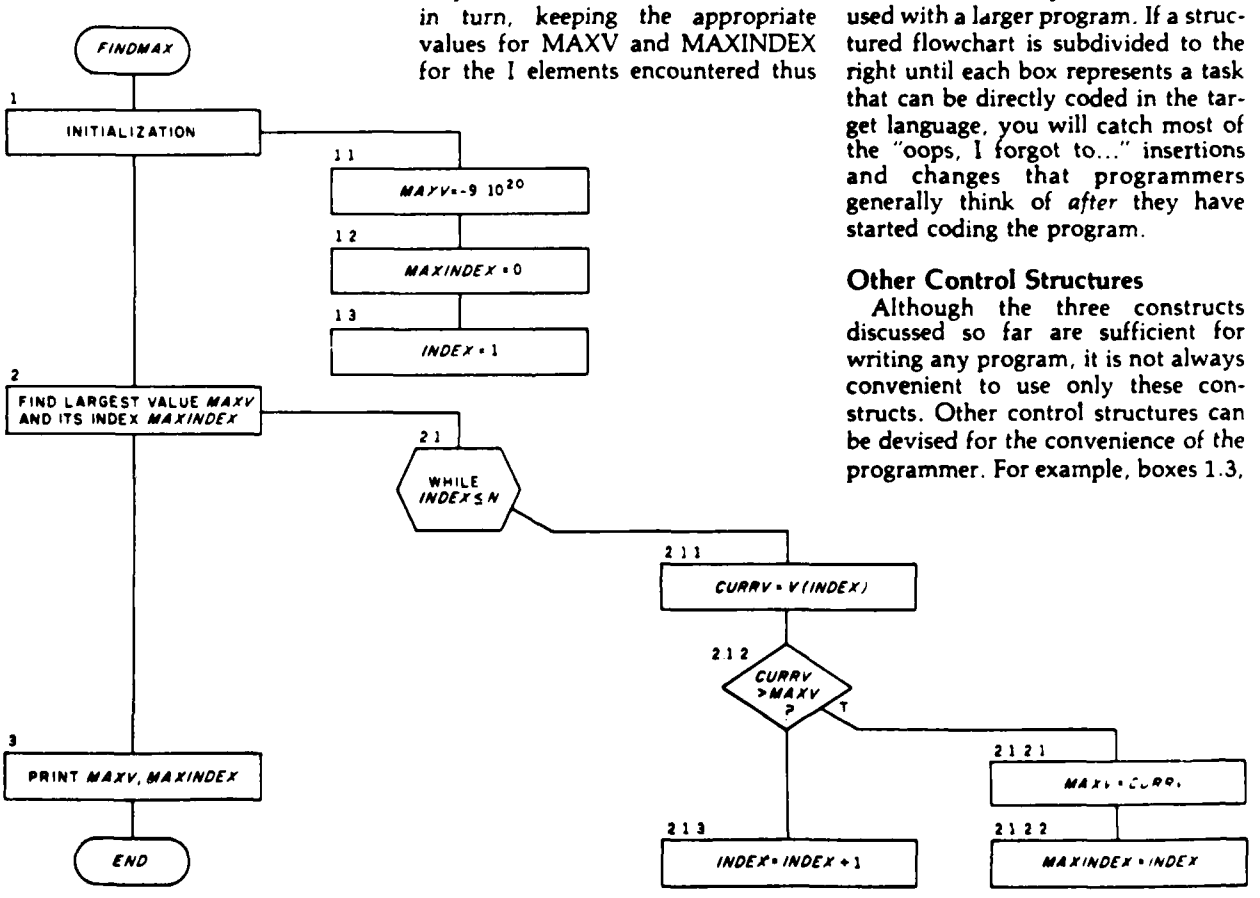


Figure 6: *Structured flowchart for program FINDMAX. Given an array V with N elements, the problem is to find the largest element, MAXV, and its index within the V array, MAXINDEX. The numbers above each box give the sequence and level of that box in relation to the entire problem. For example, box 1 can be broken into three subtask boxes: 1.1, 1.2, and 1.3.*

2.1, and 2.1.3 in figure 6 can be replaced by a control structure that is available in most programming languages—a *do-loop* that varies INDEX from 1 to N. An example of the notation I have devised for this is given in figure 7a; the body of the loop is performed according to the parameters given in the hexagon.

Another well-known control structure is the *repeat...until* loop, shown in figure 7b. The position of the body of the loop, above and to the right of its associated hexagon, is meant to signify that the body of the loop is performed *before* the condition is tested. Although the meaning of this notation does not implicitly follow from its form, it was chosen for its simplicity and consistency with the notation already developed.

Other constructs come to mind: a *case* structure, an unconditional *goto*, and two controlled *gotos*—the *restart* (restart the innermost containing loop) and the *exit* (go to the first task after the innermost containing loop). Although I have used some of these constructs for quite some time, they are not presented here because I am not yet satisfied with the notations I have developed for them. In any case, structured flowcharts are meant to be a personal notation—you should add to and modify these constructs to fit your needs.

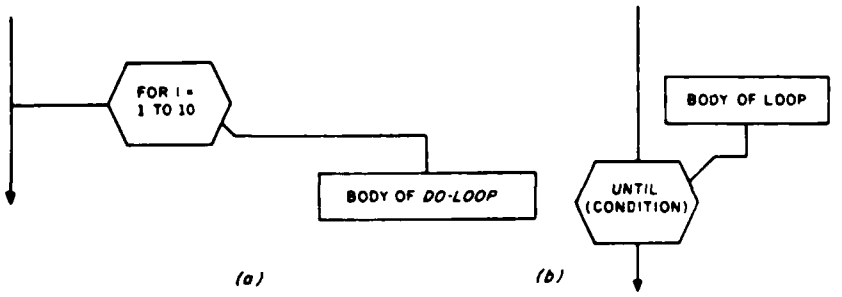


Figure 7: *Structured flowchart notation for a* do-loop *and a* repeat...until *loop. In the do-loop, figure 7a, the hexagon contains all pertinent information defining the loop, and in the form most comfortable to the user. In the* repeat...until *loop, figure 7b, the notation is interpreted as showing the body of the loop being executed before the condition is tested. In both cases, the box representing the body of the loop can be expanded to the right, into its component subtasks*

Conclusions

I have found structured flowcharts helpful in designing programs. The notation is obviously intended for weakly structured languages (like BASIC), as its utility decreases when the structure of the target language increases.

The notation is, at the moment, informal, and it should stay that way. It should be extended and modified in whatever way seems useful to you. In particular, you should use additional notation for special features of the target language (eg: global and local variables, use of a stack of intermediate computation) when applicable. If the structured flowchart is to be read by another person, however, you should define all the structures used in terms of their equivalent unstructured (conventional) flowcharts.

If the final structured flowchart is to be redrawn, you should do so with clarity in mind. Place only those boxes that help explain the overall design with the main flowchart; leave the implementation details to subordinate flowcharts.

I hope you will find this notation useful. I would appreciate your suggestions, criticism, and comments. ■

References

1. Page-Jones, Meilir. *The Practical Guide to Structured Systems Design*. New York: Yourdon Press, 1980.
2. Ross, D T, J B Goodenough, C A Irvine. "Software Engineering: Process, Principles, and Goals." *Computer*. Institute of Electrical and Electronics Engineers (IEEE), May 1975. Also *Tutorial on Software Design Techniques*. Third Edition, P Freeman and A I Wasserman, editors. Long Beach CA: IEEE Computer Society, 1980.
3. Williams, G. "Applied Structured Programming." *Program Design*, Blaise Liffick, editor. Peterborough NH: BYTE Books, 1978.
4. *Classics in Software Engineering*, E N Yourdon, editor. New York: Yourdon Press, 1979.

APPENDIX B

"What is Good Documentation?"

by

Jim Howard

What Is Good Documentation?

Jim Howard
150 Ramona Place
Camarillo CA 93010

As more and more people discover the joys of owning a microprocessor the need for good documentation will continue to grow. Information will be needed at all levels, from detailed hardware and software documentation to descriptions of which buttons to push to play your favorite game.

Who will provide this information? The simple answer is that those who know will tell those who don't know. It sounds simple, but it's not. Everywhere, complaints are made about documentation—"inadequate," "erroneous," "over my head," "bad or nonexistent," and so on. All too often, companies market excellent systems with poor or sketchy documentation, resulting in unhappy customers and unsatisfactory sales.

It's a common mistake to believe that because somebody is an expert in a subject, he can explain it to others. For example, it's assumed that a professor who knows a subject inside and out can pass on this information to students. However, whether he can or cannot depends on something else besides his knowledge of the subject. It depends on his ability to put himself in the place of the users, the students, to begin where *they* are, using their language and their knowledge level. (Of course, if there is a failure to communicate, it is the students who fail, not the professor!)

The microprocessor industry is a classic example of the communication problem. Aside from a few shining lights, microprocessor literature suffers from a bad case of "the jargons." The problem was not as serious while the technology was being pursued by only a few hobbyists, who like to work things out for themselves. Now that the public is becoming involved in large numbers, the information must adapt to the customer, not the other way around.

Aside from a few shining lights, microprocessor literature suffers from a bad case of "the jargons."

Many could undoubtedly do a better job of communicating if they followed a few principles. But doing this requires conscious dedication. And, of course, it requires principles. Those principles are what this article is about.

To translate the jargon of the expert into terms meaningful to the rest of the world, we need an *interpreter*. Such an interpreter is similar to the compiler or interpreter used in computers, which translates the source language into one the machine understands. In both cases, the source language is provided by the computer expert. The machine is the user in one case, the public in the other.

Information Design

The interpreter we require can best be referred to as *information design*. This term is better than the common term "technical writing," in that it indicates what really is required—conscious, step-by-step design. Writing is just one aspect of presenting understandable information. In fact, technical writing is similar to writing code for a computer program. If the planning and structure are sound, the writing almost takes care of itself.

There are many aspects of information design, not all of which can be

Information Design Principles

●**Content** *defines the* **breadth** *and* **depth** *of the material in a document, and is best specified by a* **topic diagram**. *Consistency and uniformity of treatment are revealed by such a diagram: One topic should not be treated in great detail and others of equal importance hardly mentioned. The breadth and depth should fit users' needs—all relevant material included, no unnecessary redundancies, and sufficient detail to allow users to understand the explanation or perform the job.*

●**Organization** *gives shape and direction. The users always know where they are, where they have been, and where they are going.* **Indexes** *and* **headings** *make the organization visible to users, so that information is located easily and quickly. Material is grouped and sequenced to flow logically and naturally from one topic to another. A* **top-down** *approach is used, to provide an overall structure before confusing users with details.* **Introductions** *and* **summaries** *tie pieces together both forward and backward, and reinforce for long-term memory.*

●**Format** *makes the information understandable through language and illustrations.* **Language** *speaks to one half of the brain—the verbal, linear side. Simple vocabulary and short, direct sentences make for ease of understanding.* **Illustrations** *speak to the other half of the brain—the nonverbal, spatial side. Illustrations are most effective when they are near the relevant text and are keyed to it through call-outs and highlights. Working together, words and illustrations present the whole "picture" as neither can alone.*

covered here. What *is* necessary is that a few key principles are made clear.

The basic objective of information design is *usability*. Whatever the user intends to do—write a program, assemble a piece of hardware, learn how a system works—the documentation must serve this purpose.

Although this may sound trivial, if you're writing a technical document, it's surprising how easy it is to lose sight of this overall requirement after page 1. The presentation can become an ego trip without your realizing it. On the other hand, it's hard to go wrong if you consistently keep the usability objective in mind.

How do we determine if a document is usable? Whatever the type of document—operator's manual, maintenance procedure, reference manual, training program—it has some *purpose*. Its purpose may be to explain a concept, describe the operation of a piece of equipment, or guide a person through an assembly procedure.

To be usable, the document must take the users from a state of incomplete knowledge about some subject to a condition of more complete knowledge. If it's a procedure, the information must guide the users through the task. In any case, the document must take them from "here" to "there."

That's what information design does: It starts where the users are and builds step by step. The information designer first asks who the users are. Then he puts himself in their place and asks, "What will they understand, with their experience? What is their technical knowledge and vocabulary? How can they best be helped?"

Next, he builds step by step. He breaks up complicated subjects into simpler parts. He leads the users gradually into new territory, helping them make their own discoveries. With each step their confidence grows and they want to learn and do more. At the end, the users know they have succeeded—and, therefore, so has the information designer.

The Elements of Information Design

If we are going to start where the users are and build step by step we need a plan of action. We need to decide:

- what information to include in the document
- how to organize it
- how to present it so it's understandable

We'll discuss these aspects under the headings of Content, Organization, and Format.

Content

The *content* of a document is the specific technical material contained in it. This should be carefully defined by boundary lines set down by the information designer.

Content really has two aspects: what information is included (*breadth*) and what is its level of detail (*depth*). A simple example will illustrate the important difference between breadth and depth: An operator's manual for a computer system might tell you to "remove and replace the printer's print wheel as necessary." The subject of print wheel replacement is thus "covered" in the manual; that is, in terms of breadth, it is part of the content. However, the lack of "how to" details may make this information of little use to many

printer users. Thus the proper *depth* of information is *not* part of the content.

A good tool to help a writer of documentation analyze breadth and depth is a *topic diagram* (figure 1), which is an arrangement of topics in boxes at different levels, with lines joining related topics. It serves a purpose similar to that of an outline, but provides an easier visual check on such elements as breadth, depth, and consistency of treatment.

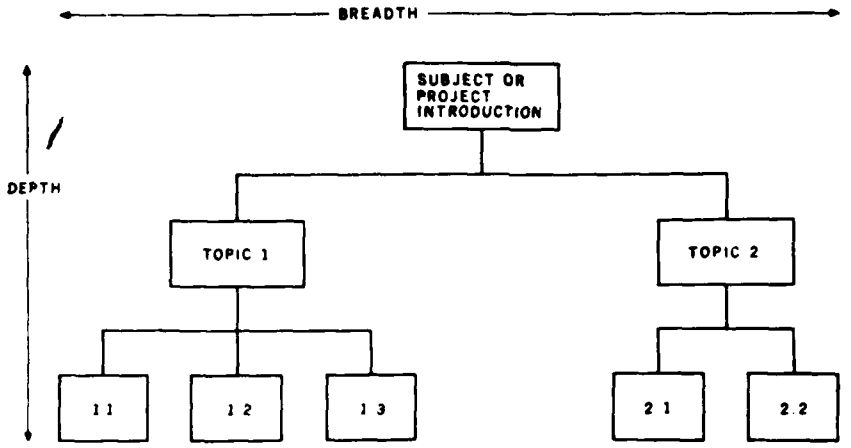


Figure 1: *A* topic diagram *is a useful tool for determining the* breadth, depth *and* consistency *of a piece of writing. Although similar in content to an outline, the topic diagram provides a clearer visual check on how topics are handled. As shown, topics 1 and 2 are major topics at the same level. Neither is a subtopic of the other and both will be treated equally when the writing is done. Subtopics represent breakdowns of each major topic. As additional topics and subtopics are added the diagram can extend downward and to the left and right*

In figure 1, topics 1 and 2 are major topics at the same level in the diagram. They might be two major components of a system, or groups of software, or procedures. Neither is a subtopic of the other and they will be treated equally in the presentation.

Subtopics are shown under each major topic: 1.1, 1.2, 1.3 under topic 1, and 2.1 and 2.2 under topic 2. These represent breakdowns of each major topic. The diagram can continue on down to further depths of subdivision and can also be extended to the left and right as additional topics are added at a given level.

We can see that the breadth of the topic diagram, particularly at the major topic level, tends to indicate the breadth of content. The depth of the diagram indicates the depth of content. While this should not be considered an infallible guide, it is useful in preliminary planning.

Another use of a topic diagram is that it gives an idea of *consistency* of coverage. A glance at figure 1 will tell the writer if topics at the same level are being treated with some consistency in how they are subdivided, or if one topic is being pursued to greater levels of detail than others. Without such a guide, it's easy to cover one topic in great detail and give other topics at the same level only token treatment or overlook them completely.

Definition of content is as important for what is *not* included as for what *is*. Many technical documents include irrelevant information. This can be particularly annoying in procedural documents, when users are trying to accomplish an exacting task. They want to get on with it, but are continually being interrupted with extraneous remarks that belong in some other part of the document or should be left out entirely.

Figure 2 shows a topic diagram for this article. As you can see, in addition to defining content, such a diagram shows a preliminary organization or structure.

Organization

To proceed step by step, we need to know where we are going and a route to get there. In other words, we need structure, or *organization*. Informa-

tion must be grouped, sequenced, and related in order to be understood. Otherwise, it is merely a jumble of disordered facts or ideas—a "shopping list." If we had to learn everything by rote memory from shopping lists, we'd be in big trouble. Once a good structure is established, all kinds of details can be hung on it and they will be understood and remembered.

Organization is also what makes information in a document easily *accessible*. Accessibility depends on both the overall structure of the document and how this structure is made visible to the user through indexing and headings. If information is organized properly, the user will be able to turn quickly to the information he wants. Once there, he will be able to continue with a minimum of routing to other parts of the document.

The importance of structure or organization can be illustrated by a very simple example—a telephone book. Have you ever stopped to think how useless a telephone book would be if the names were listed randomly rather than alphabetically? The important aspects of structure or organization include indexing and headings, grouping and sequencing, routing, and introductions and reviews.

Indexing and Headings

Indexing and headings are the means by which the organization of the document is made easily visible to users. A writer may actually have a good organization, but if it is not clear to users, it will not really have served its purpose.

Indexing as used here includes both the standard type of index found at the end of a document and the table of contents. The index should be set up with the idea that users will sometimes look for items alphabetically, as in a dictionary. Many items that are too small or too specific to be included in the table of contents are made accessible with a good index.

Often a table of contents can be usefully constructed in two parts: an overall table in front and more detailed tables with each major section of the document. This avoids an unwieldy table up front. Figure 3 provides an example of a two-part table of contents. The main table (on the left in the figure) would appear in the front of the document. Each major section would start with its own table of contents (on the right in figure 3) showing the more detailed headings and subheadings in the section.

A consistent set of *headings* serves to make information accessible. Headings also help users remember

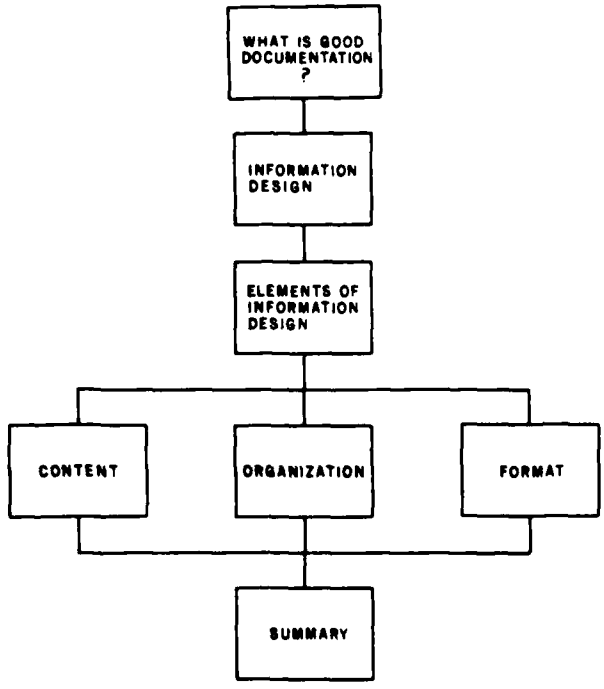


Figure 2: *A topic diagram written for this article.*

where they are, which is just as important. Thus high-level headings should be repeated frequently, for example as a running head at the top of each page. Having the relevant headings always in front of the user makes the structure visible, and details are then assimilated more easily.

Grouping and Sequencing

The overall organization of the document is established by how the content material is grouped and sequenced. Again, the topic diagram is useful during the planning stages in making visible the planned organization of the document.

Whether the document is procedural or descriptive, grouping of the topics should be based on a logical pattern and the relevance of different items. For example, procedural tasks normally performed together (such as the various steps required to start up a computer system) should be grouped together. In a system description, the individual descriptions of system components would normally be grouped together, as in the example table of contents shown in figure 3.

Sequencing is one of the most critical parts of the structure. The user is being led step by step from the known to the unknown, from the simple to the complex. Here the *top-down* structuring principle frequently used in writing computer programs also applies. The sequence should begin at the top and give the readers the big picture before engulfing them with details. It is not unusual to begin reading a document and find yourself up to your ears in technical details before you really know what's going on.

Most equipment operations and human activities have a natural or normal sequence that should be preserved in the documentation. For example, you normally gather together all the tools and supplies required for an activity before starting; therefore, this information should logically precede the activity description. It is disconcerting to have to stop in the middle of a task and run to the hardware store to buy some item.

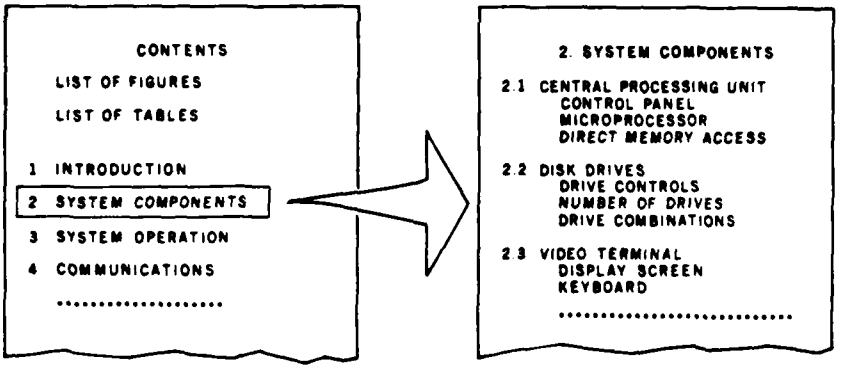


Figure 3: *An example of a two-part table of contents. By using an overall table in the front of the document, and a more detailed table later, an initial unwieldy table is avoided where a user would be subjected to unwanted detail.*

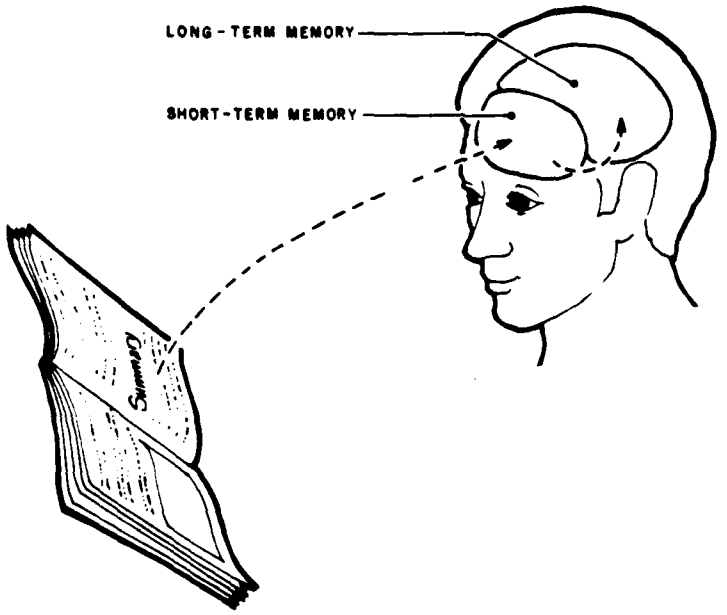


Figure 4: *Summaries and long-term memory. In the human brain, memory is divided into short-term memory and long-term memory. Although the capacity of long-term memory is large, all information must first pass through a short-term memory. When writing, the inclusion of summaries, reviews, and question-and-answer sections is an effective way of passing information into long-term memory.*

Routing

Once you start using a document it is inconvenient to have to refer to other parts of the document, or to other documents. The more often you are routed, and the more pages you have to thumb through to get there, the less useful the document. On the other hand, if all information is repeated at each point of need, a bulky document can result. Obviously, judgment is required in weighing these trade-offs. For example, you wouldn't want to tell a user how to solder a particular type of joint every time it came up—you would set aside a special section for this purpose. However, if a safety precaution applies to a number of different tasks in the document, it is better to accept the redundancy and repeat the precaution.

Introductions and Reviews

A general rule is to prepare users for what is coming and to remind them of where they have been. Proceeding through a document, users may forget where they are, forget what has gone before—and decide they didn't really want to learn this anyway. Information should be

designed to help users relate backward and forward and recognize and retain key points along the way.

Further, readers need introductory instructions to help them find and use information. For example, the numbering schemes for tasks or illustrations, the use of safety symbols, notes, cautions, and warnings, and the treatment of information about tools and supplies should be briefly explained. If these instructions are backed up by consistent information presentation (see **Format** section), users will quickly learn what to expect, no matter where they are in the document.

Simple reviews at key points reinforce information and help users retain it in memory. Human memory, to put it simply, consists of two parts, "short-term" and "long-term." Whereas capacity is very limited in STM (short-term memory), the capacity of LTM (long-term memory) is large indeed. The catch is that information can get to LTM only through STM. Summaries and reviews and question-and-answer sessions are effective ways of establishing information firmly in LTM. This important concept is illustrated in figure 4.

Format

Format usually has the rather narrow meaning of "physical layout of the page." Here the term is meant also to include the rules that govern text and illustrations—that is, how information is presented on a page.

The general rule is that language and illustrations should work together. Each is an effective way of presenting certain kinds of information, and relatively ineffective for other kinds. When combined properly, they form a powerful presentation technique.

People will readily admit that pictures can do things that words cannot and vice versa. And yet it is surprising how often we find ourselves reading words, words, words, when a visual or two would have helped the presentation considerably. Many ideas become clearer with an illustra-

tion, and some kinds of information can hardly be communicated at all without one. If you want to tell someone what something looks like, show a diagram or a photograph.

It is known that the left and right sides of the brain are quite different. For most people, the left side is dominant and works mostly with linear, sequential logic (like a computer). It is also the verbal side and controls *language*.

The right side specializes in images, music, *pictures*—it deals in spatial and visual concepts, in contrast to the linear, verbal left brain. Schools, with their traditional emphasis on verbal skills, have tended to neglect the right side of the brain. People who are less adept with their left brain have suffered as a result. Einstein, for example, was a poor student in language, but had a great ability to visualize (see figure 5).

The ideal combination is words and pictures working together, each doing what it does best. In a procedure, for example, words can tell readers what to do and how to do it; pictures can tell them what it looks

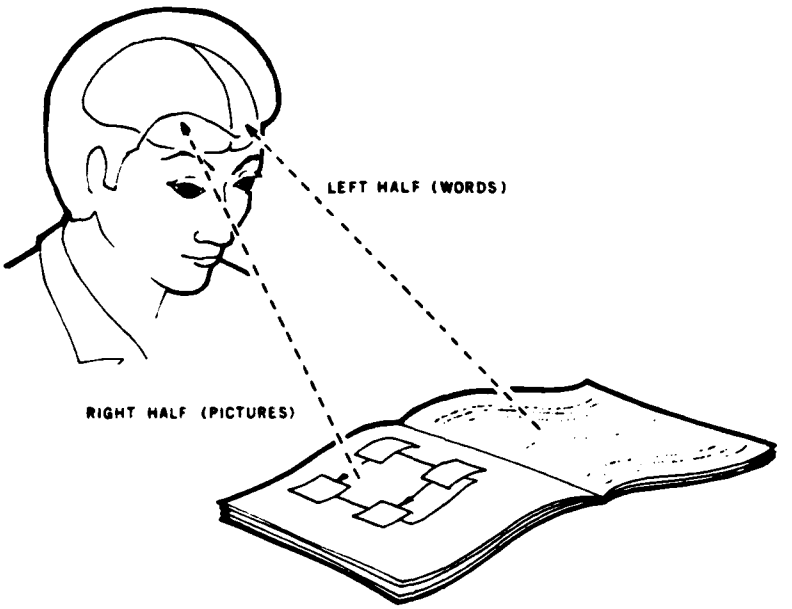


Figure 5: *The left and right sides of the human brain are very different. In most humans, the left side, which works mostly with linear and sequential logic, is dominant. The left side also controls verbal communications. The right side of the brain deals in spatial, visual, and more holistic concepts. One of the best ways of imparting information to the reader is through a combination of both words and pictures, thus enabling the reader to use both sides of the brain.*

like and where it is. For descriptive material, words and diagrams will do a good job of explaining and describing, *provided* they are working together. When you decide to use pictures to communicate with readers, follow the flow through step by step. Don't be content with offering an occasional "amazement diagram" and a "see figure so-and-so." You can perhaps wake up the right half of the reader's brain this way, but to get it working with the left half as a unit—whole-brain learning—make the words and pictures work together.

Here are some guidelines on how to do this, discussed under the following headings: keying text to illustrations, positioning text and illustrations, and limiting information density.

Keying Text to Illustrations

The mutual reinforcement of text and illustrations can be strengthened by keying the text to the illustration. This can be done by a liberal use of highlights and call-outs, which are "talked to" in the text.

For complicated diagrams, an indexing system can be used. An example of this common technique is shown in figure 6. Three parts of an electrical unit are designated A, B, and C in the picture on the right. These same letters are used in the text on the left to refer to these specific parts. This method can be used with fairly complex diagrams without confusing the reader. The alphabetical or numerical symbols take up little room on the diagram and can be ordered (for example, clockwise in figure 6) to make it easy to locate any symbol.

Highlights and call-outs help the user zero in on the main items of interest in a picture. A heavy outline or shading or color, together with a call-out of the item of interest, can make the text and illustration mutually support each other and help the user relate illustration to text.

Consistent, standard nomenclature should be used in linking text to illustration, and indeed throughout the document. Information becomes less accessible and less understandable if the same item is referred to by different names.

Positioning Text and Illustrations

Because the text and related pictures should work together, they should be positioned close together. Ideally, the user should be able to work back and forth between text and illustration without having to turn a page. While this ideal is sometimes impractical, it is usually possible to keep the illustration close to the relevant text. For important, frequently referenced figures, fold-outs are sometimes the answer.

Limiting Information Density

Information is like food. If readers eat too fast, or too much at one time, they get indigestion. If information is presented too fast or in too large doses, readers will get confused. This is because of the limited capacity of short-term memory. Therefore, like food, information must be broken up into "bite-size" pieces to be digestible.

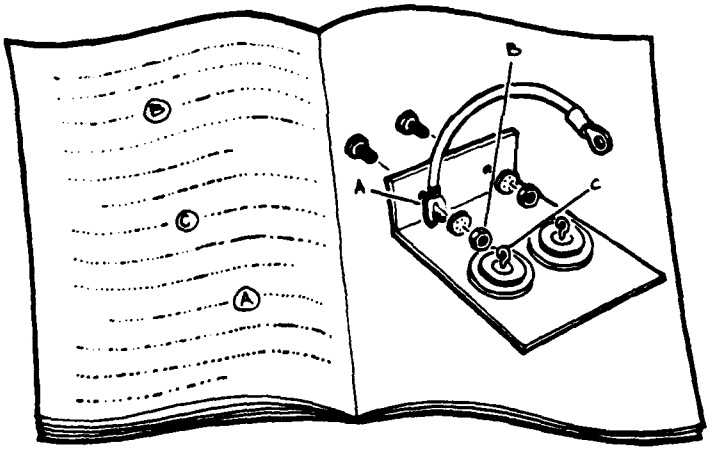


Figure 6: *Keying text to illustrations. The mutual reinforcement of text and illustrations (as shown in figure 5) can be strengthened by keying the text to the illustrations through the use of highlights and call-outs which are "talked to" in the text.*

Good format does this.

Language should be simple and direct. Only words the reader understands should be used, with new words explained as they are introduced. Explanations are easier to read and understand if sentences are short and simple, and if words have few syllables.

Illustrations should not be cluttered with unnecessary information. If they are too "busy," pictures become confusing and are less useful. To avoid a profusion of details, illustrations can be used in a progression from simple to more complex. This is related to top-down sequencing. An initial overall figure can give the "big picture," which is easy to understand and serves as a beginning structure for proceeding to more detailed illustrations. In forming such progressions, it's important to preserve the relative locations of the parts of whatever is being pictured. For example, if a simple block diagram of a microprocessor leads off the series, subsequent more detailed diagrams and schematics should show the various parts of the blocks in the same relative positions as the original block. An example is shown in figure 7. Note that the lower detailed diagram preserves the relative positions, established by the upper figure, of the major parts of the system.

Earlier we said that microprocessor literature is suffering from a bad case of "the jargons." However, you'll see by now that there is much more to good documentation than avoiding jargon. You probably have had the experience of reading something and finding that it was very difficult to *follow, even though you seemed to* understand all the words. In this case, the author managed to avoid technical terminology but failed in other important areas. Good technical documentation requires a highly disciplined approach, and that approach is provided by information design. Those who adopt a go-as-you-please approach may score a success now and then, but it will be by accident. They have no way of knowing whether they have really reached their audience. In many cases they have not. ■

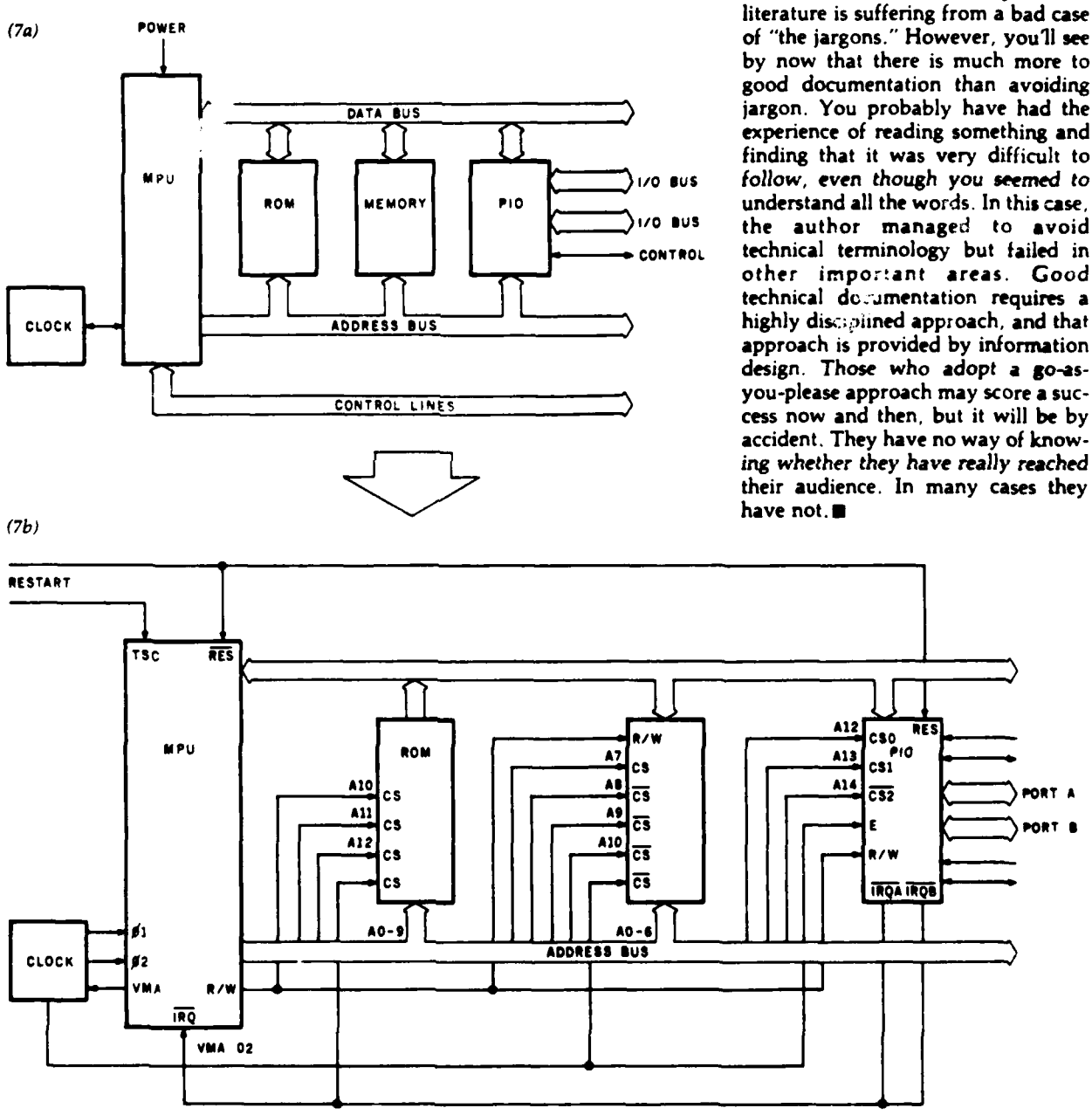


Figure 7: *To avoid reader confusion, illustrations should be used in a progression from less detail to more. An initial block diagram (7a) can give the overall picture before going into greater detail (7b). When forming these progressions, it's important to keep parts in the same relative positions.*